



Git i GitHub

Kontrola wersji, zarządzanie projektami
i zasady pracy zespołowej

Mariot Tsitoara

Tytuł oryginału: Beginning Git and GitHub: A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer

Tłumaczenie: Maksymilian Gutowski

ISBN: 978-83-283-8735-5

First published in English under the title Beginning Git and GitHub: A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer by Mariot Tsitoara, edition: 1

Copyright © 2020 by Mariot Tsitoara

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature. APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

Polish edition copyright © 2022 by Helion S.A.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<https://helion.pl/user/opinie/wprgit>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)



Spis treści

O autorze	11
O korektorze merytorycznym	12
Podziękowania	13
Wstęp	14
Część I	
Kontrola wersji z Gitem	15
Rozdział 1. Systemy kontroli wersji	17
Czym jest kontrola wersji?	17
Dlaczego potrzebujesz takiego systemu?	18
Jaki mamy wybór?	20
Lokalne systemy kontroli wersji	20
Scentralizowane systemy kontroli wersji	21
Rozproszone systemy kontroli wersji	22
Czym jest Git?	23
Co Git potrafi?	24
Jak działa Git?	25
Jak przebiega praca w Gicie?	26
Podsumowanie	28
Rozdział 2. Instalacja i konfiguracja	30
Instalacja	30
Windows	30
Mac	37
Linux	39
Konfiguracja Gita	39
Podsumowanie	41

Rozdział 3. Początki	42
Repozytoria	42
Katalog roboczy	44
Przechowalnia	46
Commity	47
Z Gitem na głęboką wodę	50
Podsumowanie	51
Rozdział 4. Z Gitem na głęboką wodę	52
Ignorowanie plików	52
Przeglądanie logów i historii zmian	57
Przeglądanie poprzednich wersji	59
Przeglądanie aktualnych zmian	60
Podsumowanie	62
Rozdział 5. Commity	63
Trzy stany Gita	63
Poruszanie się po wersjach	64
Cofanie commita	66
Modyfikowanie commita	69
Poprawianie commita	73
Podsumowanie	74
Rozdział 6. Git — najlepsze sposoby postępowania	75
Opisy commitów	75
Zatwierdzanie — najlepsze praktyki	76
Co należy robić?	77
Czego nie należy robić?	78
Jak działa Git — powtórka	79
Podsumowanie	81
Rozdział 7. Zdalny Git	82
Dlaczego repozytoria zdalne?	82
Jak to działa?	83
Pójdźmy na łatwiznę	84
Podsumowanie	86
Część II Zarządzanie projektami z GitHubem	87
Rozdział 8. GitHub — podstawy	89
GitHub — przegląd	89
GitHub a open source	90
Użytek osobisty	94

GitHub dla przedsiębiorstw	95
Podsumowanie	96
Rozdział 9. GitHub — szybki start	97
Zarządzanie projektem	97
Jak działają repozytoria zdalne?	99
Łączenie repozytoriów	101
Wypychanie do repozytoriów zdalnych	102
Podsumowanie	106
Rozdział 10. Podstawy zarządzania projektami — zgłoszenia	108
Czym są zgłoszenia	108
Wydanie zgłoszenia	109
Interakcja ze zgłoszeniem	113
Etykiety	113
Przydziały	117
Łączenie zgłoszeń z commitami	118
Praca nad commitem	118
Odwołania do zgłoszeń	119
Zamykanie zgłoszenia przy użyciu słów kluczowych	121
Podsumowanie	124
Rozdział 11. Zarządzanie projektami na poważnie — gałęzie	125
Obieg pracy na GitHubie	125
Gałęzie	127
Tworzenie gałęzi	129
Przełączanie się na inną gałąź	129
Usuwanie gałęzi	131
Scalanie gałęzi	133
Wypychanie gałęzi do zdalnego repozytorium	136
Podsumowanie	138
Rozdział 12. Sprawniejsze zarządzanie projektami — pull requesty	139
Dlaczego należy używać pull requestów?	139
Pull requesty — przegląd	140
Ściąganie	140
Co robi pull request?	141
Tworzenie pull requesta	142
Przeglądy kodu	148
Przeprowadź przegląd kodu	148
Zostaw komentarz	149
Aktualizowanie pull requesta	152
Podsumowanie	155

Część III	Praca zespołowa w Gicie	157
Rozdział 13.	Konflikty	159
	Jak działa scalanie?	159
	Ściąganie	160
	Scalanie fast-forward	161
	Konflikty scalania	165
	Ściąganie commitów z origin	167
	Rozwiązywanie konfliktów scalania	171
	Podsumowanie	177
Rozdział 14.	Więcej o konfliktach	178
	Wypychanie po rozstrzygnięciu konfliktu	178
	Przeglądanie zmian przed scaleniem	179
	Sprawdź lokalizację gałęzi	179
	Przejrzyj różnice na gałęziach	179
	Zrozumieć scalanie	180
	Ograniczanie konfliktów	181
	Dobry obieg pracy	181
	Przerwanie scalania	182
	Wizualne narzędzia Gita	182
	Podsumowanie	182
Rozdział 15.	Narzędzia GUI Gita	183
	Domyślne narzędzia	183
	Commitowanie: Git GUI	183
	Przeglądanie: gitk	192
	Narzędzia IDE	192
	Visual Studio Code	193
	Atom	195
	Specjalistyczne narzędzia	195
	GitHub Desktop	195
	GitKraken	196
	Podsumowanie	196
Rozdział 16.	Zaawansowane polecenia Gita	198
	Cofanie zmian	198
	Składowanie	199
	Resetowanie	203
	Podsumowanie	205

Część IV	Dodatkowe zasoby	207
Rozdział 17.	Więcej o GitHubie	209
	Wiki	209
	Strony GitHuba	211
	Wydania	214
	Tablice projektowe	214
	Podsumowanie	218
Rozdział 18.	Praca z Gitem — częste problemy	220
	Repozytorium	220
	Zaczynanie od nowa	220
	Zmiana repozytorium origin	221
	Katalog roboczy	222
	Git diff jest pusty	222
	Cofanie zmian w pliku	222
	Commity	222
	Błąd w commicie	223
	Cofanie commitów	223
	Gałęzie	224
	Odlączony HEAD	224
	Praca na niewłaściwej gałęzi	225
	Pobieranie zmian z gałęzi rodzica	225
	Rozbieżność gałęzi	226
	Podsumowanie	227
Rozdział 19.	Git i GitHub — obieg pracy	229
	Jak korzystać z tego obiegu pracy?	229
	Obieg pracy z GitHubem	230
	Každy projekt zaczyna się od projektu na GitHubie	230
	Každy działanie zaczyna się od zgłoszenia	230
	Nie wolno wypychać zmian bezpośrednio na gałąź główną	230
	Každy scalenie z gałęzią główną wymaga PR-a	231
	Dokumentuj kod w wiki	231
	Obieg pracy z Gitem	231
	Orientuj się, gdzie się znajdujesz	231
	Ściągaj zmiany z repozytorium zdalnego	231
	Dbaj o opisy commitów	232
	Nie zmieniaj historii	232
	Podsumowanie	232

ROZDZIAŁ 5.



Commity

W poprzednim rozdziale omówiłem podstawowe funkcje Gita. Wiesz już, jak sprawdzić historię i przejrzeć zmiany wprowadzone w aktualnej wersji. Commity Gita są jednak twardym orzechem do zgryzienia, więc w tym rozdziale opowiem o nich więcej. Zaczniemy ponownie od tajników działania Gita i zaznajomienia się z terminologią, a następnie wyjaśnię, jak wywoływać i sprawdzać wcześniejsze wersje. Zaczynamy!

Trzy stany Gita

Zanim opowiem o szczegółach, wróćmy do podstaw i przypomnijmy sobie, jak Git działa. Z całą pewnością pamiętasz trzy stany, w jakich może się znajdować plik. Jeśli nie pamiętasz, nie pomijaj tego rozdziału. Ta wiedza jest nieodzowna w pracy z Gitem. Jeśli natomiast pamiętasz te stany, to także nie pomijaj tego rozdziału — poświęciłem naprawdę dużo czasu, by go napisać.

Jak pokazałem w poprzednim rozdziale, nie wszystkie pliki są śledzone w Gicie; niektóre z nich można ignorować za pomocą pliku `.gitignore`. Są też pliki, które nie są wprawdzie ignorowane, ale nie są też jeszcze śledzone przez Gita. Mowa o nowo utworzonych plikach, które nie zostały ujęte w migawce (czy też commicie).

Śledzone pliki mogą się znajdować w jednym z trzech stanów:

- Zmodyfikowany — do pliku zostały wprowadzone zmiany.
- Dodany do przechowalni — do pliku zostały wprowadzone zmiany i został on przygotowany do uchwycenia w migawce.
- Zatwierdzony — została zachowana migawka całego projektu i plik został w niej uchwycony.

Nieśledzone pliki pozostają takimi, dopóki nie wyślesz ich do przechowalni i nie zatwierdzisz albo dopóki nie polecisz, aby były ignorowane.

Pamiętaj: Git nie rejestruje zmian, tylko migawki. Przy każdym commicie zapisany zostaje stan całego projektu, a nie wszystkie drobne zmiany, które wprowadzono.

Ciekawostka: Git działa szybko, ponieważ zawsze pracujesz na ostatnim zapisanym stanie projektu. Kiedy chcesz zobaczyć wcześniejszy commit, Git pokazuje stan projektu z określonej daty. Wiele systemów kontroli wersji zapisywało wszystkie wprowadzane zmiany, więc kiedy programista chciał przywrócić poprzedni stan, systemy te wprowadzały zmiany wstecz. Gdy projekt się rozrasta, taki sposób działania może być przyczyną wielu problemów z szybkością i pamięcią. Czy sposób, w jaki Git funkcjonuje, nie prowadzi do tworzenia olbrzymich baz danych? Nie, ponieważ kiedy wykonujesz migawkę, pliki, które nie uległy zmianie, nie zostają zapisane ponownie, lecz jest tworzony odnośnik do pliku.

Wróćmy jeszcze raz do tematu trzech stanów i spójrzmy na łączące je zależności:

- Pracujesz na katalogu roboczym. Jest to po prostu katalog, który utworzyłeś przed zainicjalizowaniem repozytorium. To właśnie w nim będziesz odczytywać i edytować pliki.
- Przechowalnia jest miejscem, do którego trafiają zmienione pliki przed utworzeniem migawki całego projektu. Nie możesz utworzyć migawki, jeśli pliki nie zostały wysłane do przechowalni. Jedynie pliki znajdujące się w przechowalni (i niezmienione pliki) zostaną uwzględnione w migawce. Pliki poza przechowalnią (śledzone lub nie) oraz ignorowane zachowają swój stan.
- Baza danych lub katalog `.git` przechowuje każdą wykonaną migawkę. Takie migawki nazywa się commitami.

Pamiętaj: przechowywanie dotyczy jedynie wybranych przez Ciebie zmienionych plików, a zatwierdzanie dotyczy całego projektu. Plik przechowujesz, a projekt commitujesz.

Poruszanie się po wersjach

Często będzie Ci zależało nie tylko na możliwości sprawdzenia zmian, jakie wprowadzono do projektu, ale także na przyjrzeniu się stanowi, w jakim się znajdował — zobaczeniu wykonanej migawki. W Gicie to wszystko jest proste.

Kiedy chcesz wywołać wcześniejszy stan projektu w katalogu roboczym, musisz przełączyć się na commit poleceniem `git checkout`. Ponieważ w wyniku tego pliki znajdujące się w katalogu roboczym ulegną zmianie, musisz się upewnić, czy nie znajdują się w nim przypadkiem pliki, które nie zostały wysłane do przechowalni. Pliki nieśledzone mogą wciąż się w nim znajdować, ponieważ Git jeszcze nie rejestruje ich stanu.

Aby obejrzeć migawkę projektu, wydajemy polecenie `git checkout` wraz z nazwą commita jako parametrem:

```
$ git checkout <nazwa>
```

Wypróbujmy to! Otwórz swój bieżący projekt w edytorze tekstu i spójrz na jego zawartość. Sprawdź teraz poprzedni commit tak jak na rysunku 5.1.

■ **Ostrzeżenie** Nie możesz przełączyć się na inny commit, jeśli katalog roboczy nie jest pusty. Koniecznie zatwierdź wszystkie zmiany przed przełączeniem się na inną migawkę.

Uważaj, by przypadkiem niczego nie zmienić przy sprawdzaniu wcześniejszych commitów. To działa jak na filmach: lepiej nie zmieniać przeszłości!

```

MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git log
commit 8ba74a5546782e38d1c2d6dafd2386e814034c69 (HEAD -> master)
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Mon May 27 21:31:51 2019 +0200

    Rearrange .gitignore

commit b2eccbf5b54c0f5b6d34b2432245a1a582a96f6
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 21:20:51 2019 +0200

    Add .gitignore

commit 5f57824bdc7b704d17e8a9cbf36146f43eb0269a
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:18:12 2019 +0200

    Finish task 1: mittens

commit 9f180aae6d70f83a5252b0d1be2d68321f5b2146
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:17:11 2019 +0200

    Doing task 1: mittens

commit 1c3f05747ab8a5416d1be8efbbd3865206681275
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:15:26 2019 +0200

    Create TODO

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git checkout 9f180aa
Note: checking out '9f180aa'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b <new-branch-name>

HEAD is now at 9f180aa Doing task 1: mittens
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo ((9f180aa...))
$ |

```

Rysunek 5.1. Sprawdzanie poprzednich commitów

W edytorze tekstu zobaczysz, że projekt wygląda teraz dokładnie tak jak wtedy, kiedy wykonywałeś migawkę. Właśnie to jest najlepsze w Gicie. Nic, co zostało uchwycone w migawce, nie może zostać utracone!

Zapoznajmy się teraz z odrobiną terminologii Gita. Pierwsze pojęcie to „head”, który jest po prostu wskaźnikiem do commita. Kiedy mówimy o commitach, nie nawiązujemy do nazwy, lecz do heada.

Gdy przełączamy się między różnymi commitami, musimy jakoś się orientować, na którym headzie jesteśmy. Aktualny head (ten, który przeglądamy) oznaczany jest po prostu jako *HEAD*.

I już! Head jest wskaźnikiem odsyłającym do commita (w repozytorium może się znajdować wiele headów), a head wskazujący na aktualnie przeglądany commit to *HEAD*.

Ćwiczenie: poruszanie się po historii

Przełącz się z jednego commita na inny poleceniem `git checkout`. Pod żadnym pozorem niczego nie zmieniaj.

Jak jednak wrócić do normalnego, bieżącego katalogu roboczego? Ponieważ nie wprowadziliśmy żadnej dużej zmiany do repozytorium, powrót do katalogu roboczego sprowadza się do przełączenia się na jedyną gałąź, jaką mamy. Zgodnie z konwencją nosi ona nazwę *master*:

```
$ git checkout master
```

Wypróbuj! Pamiętaj jednak o świętych zasadach podróży w czasie:

- Wolno cofać się w przeszłość tylko wówczas, jeśli w teraźniejszości panuje porządek (czyli nie ma żadnych niewysłanych do przechowalni plików w katalogu roboczym).
- Nie wolno zmieniać przeszłości (przynajmniej dopóki nie będziesz mieć większego doświadczenia).

Nie zapomnij o przełączeniu się na aktualną gałąź (*master*) po przeglądaniu starszych wersji.

Cofanie commita

Nadejdzie czas, kiedy wyślesz pliki do przechowalni i je zatwierdzisz, aby później się rozmyślić. Każdemu może się to zdarzyć. W przypadku tradycyjnych metod (bez wersjonowania) cofanie zmian jest bardzo trudne, zwłaszcza kiedy ma się do czynienia ze zmianami wprowadzonymi dawno temu. W Gicie wystarczy jednak użyć jednego polecenia: `git revert`.

Czy nie lepiej byłoby usunąć commit? Nie, ponieważ obowiązuje zasada podróży w czasie z poprzedniego punktu: nie wolno zmieniać przeszłości. Zatwierdzone zmiany muszą takimi pozostać dla dobra historii; zmiana tego, co stało się w przeszłości, jest bardzo niebezpieczna i w dodatku niezgodna z intuicją. Zamiast tego należy użyć polecenia `git revert` do utworzenia nowego commita, będącego przeciwieństwem commita, który chcesz cofnąć.

Cofnięcie commita jest zatem zatwierdzeniem jego przeciwieństwa. To takie proste! Aby użyć tego polecenia, musisz podać nazwę commita jako parametr:

```
$ git revert <nazwa commita>
```

Możesz cofnąć dowolny commit. Musisz jedynie się upewnić, że pracujesz na czystym katalogu roboczym. Nie zapomnij zatem o wysłaniu plików do przechowalni i zatwierdzeniu ich przed cofnięciem commita. Wypróbujmy to!

Na początek musisz sprawdzić, czy katalog roboczy jest czysty (rysunek 5.2).

```

MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git status
On branch master
nothing to commit, working tree clean

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |

```

Rysunek 5.2. Sprawdzenie katalogu roboczego poleceniem git status

Do perfekcji. Skoro już wiemy, że katalog roboczy jest czysty, nadszedł czas, aby przejrzeć historię i zorientować się, który commit należy cofnąć. Powinniśmy otrzymać rezultat taki jak na rysunku 5.3.

```

MINGW64:/c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git log
commit 8ba74a5546782e38d1c2d6dafd2386e814034c69 (HEAD -> master)
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Mon May 27 21:31:51 2019 +0200

    Rearrange .gitignore

commit b2eccfbf5b54c0f5b6d34b2432245a1a582a96f6
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 21:20:51 2019 +0200

    Add .gitignore

commit 5f57824bdc7b704d17e8a9cbf36146f43eb0269a
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:18:12 2019 +0200

    Finish task 1: mittens

commit 9f180aae6d70f83a5252b0d1be2d68321f5b2146
Author: Mariot Tsitoara <mariot.tsitoara@gmail.com>
Date: Thu May 23 20:17:11 2019 +0200

```

Rysunek 5.3. Sprawdzenie historii commitów poleceniem git log

-
- **Uwaga** Jeśli nie podoba Ci się format, w jakim przedstawiona jest historia commitów, możesz podać parametr `--one-line`, aby ograniczyć ilość wyświetlanych informacji. Przykład widać na rysunku 5.4.
-

Po zapisaniu opisu commita (tak jak zwykle) otrzymujesz streszczenie zawartości migawki. Rysunek 5.6 przedstawia wynik, jaki uzyskasz po wykonaniu poleceń i zapisaniu opisu commita.

```

MINGW64; c:/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git status
On branch master
nothing to commit, working tree clean

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git log --oneline
8ba74a5 (HEAD -> master) Rearrange .gitignore
b2eccfb Add .gitignore
5f57824 Finish task 1: mittens
9f180aa Doing task 1: mittens
1c3f057 Create TODO

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git revert 5f57824
[master 2b7e227] Revert "Finish task 1: mittens"
2 files changed, 1 insertion(+), 1 deletion(-)
delete mode 100644 DONE.txt

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |

```

Rysunek 5.6. Streszczenie cofnięcia zmian

Jak widzisz, cofanie zmian jest w Gicie bardzo proste. Należy pamiętać, że `git revert` tworzy jedynie nowy commit ze zmianami przeciwnymi do tych, które wprowadzono wcześniej. Oznacza to, że możesz też cofnąć cofnięcie! Cofnięcie cofnięcia polega na ponownym zatwierdzeniu pierwotnego commita, a oba cofnięcia wzajemnie się neutralizują. Same commity jednak pozostaną w historii, ponieważ nie możesz zmienić przeszłości.

-
- **Uwaga** Tak naprawdę możesz zmienić przeszłość. Nigdy jednak tego nie rób. To bardzo zły pomysł, którego skutkiem jest masa problemów.
-

Modyfikowanie commita

Jak obiecałem Ci w poprzednim rozdziale, dowiesz się teraz, jak zmodyfikować commit. Możesz to robić, jeśli zapomniałeś o wysłaniu pliku do przechowalni lub jeśli chcesz zmienić opis commita. Nie powinno się modyfikować wielu plików, jako że działanie tej funkcji jest mało intuicyjne. W kolejnym rozdziale szczegółowo omówię, kiedy i gdzie należy to robić. Powtarzam po raz kolejny: pod żadnym pozorem nie próbuj zmieniać przeszłości.

Aby zmodyfikować commit, musisz użyć polecenia `git commit` z parametrem `--amend`. Otworzysz w ten sposób domyślny edytor tekstu tak jak w przypadku zwykłego commita, z tym że wysłane do przechowalni pliki i opis commita już tam będą.

```
$ git commit --amend
```

Wystarczy następnie zapisać plik i zamknąć edytor tekstu, tak jak zwykle. Pojęcie „modyfikacja” jest nieco mylące, ponieważ nie modyfikujesz commita, tylko tworzysz nowy i zastępujesz aktualny. Od tej pory będę zatem używał słowa „poprawianie”.

Poprawienie commita polega na wzięciu wszystkiego z przechowalni i utworzeniu nowego commita. Jeśli zatem chcesz dodać nowy plik do commita lub jakiś z niego usunąć, możesz w tym celu do woli poprzenosić pliki z przechowalni i do niej. Pamiętaj: aby wycofać plik z przechowalni, musisz użyć polecenia `git reset HEAD <nazwa_pliku>`. Poniżej zobaczysz przykład.

Sięgnijmy ponownie po swoją listę zadań do wykonania. Edytuj istniejący plik, a następnie utwórz dwa nowe pliki o nazwach *filenottocommit.txt* i *fileforgotten.txt* tak jak na rysunku 5.7.

Name	Date modified	Type	Size
another	2019-05-27 22:20	File folder	
build	2019-05-23 21:57	File folder	
folder	2019-05-23 22:08	File folder	
.gitignore	2019-05-27 22:20	Text Document	1 KB
DOING.txt	2019-06-17 23:11	Text Document	1 KB
DONE.txt	2019-08-12 23:45	Text Document	0 KB
fileforgotten.txt	2019-08-12 23:46	Text Document	0 KB
filenottocommit.txt	2019-08-12 23:46	Text Document	0 KB
PRIVATE.txt	2019-05-27 22:20	Text Document	0 KB
TODO.txt	2019-06-17 23:31	Text Document	1 KB

Rysunek 5.7. Wszystkie pliki w katalogu roboczym

Możesz sprawdzić aktualny stan projektu poleceniem `git status`:

```
$ git status
```

W zależności od tego, ile plików dodałeś do tej pory do projektu, możesz uzyskać nieco inny rezultat, ale wciąż będzie podobny do tego z rysunku 5.8.

Kolejną czynnością, jaką należy wykonać, jest wysłanie plików do przechowalni, aby włączyć je do commita. Podaj nazwy zmienionych plików i *filenottocommit.txt*:

```
$ git add TODO.txt DONE.txt filenottocommit.txt
```

Z ostatniego rozdziału wiesz, że przed zatwierdzeniem zawsze trzeba sprawdzać, co wysłaliśmy do przechowalni, używając polecenia `git diff --staged`. Wyobraźmy sobie jednak, że zapomnieliśmy to zrobić i od razu zatwierdziliśmy commit.

```
$ git commit
```

W takim przypadku trafimy na komunikat opisujący zmiany do zatwierdzenia (rysunek 5.9).

Jak widzisz, wyróżnione są pliki do zatwierdzenia i nieśledzone. Dość trudno ich nie zauważyć, ale udajmy, że je przeoczyliśmy. Wpisz prosty opis commita, zapisz plik i zamknij edytor. Zobaczysz standardowe streszczenie widoczne na rysunku 5.10.

W streszczeniu commita widzisz, że zatwierdziłeś niewłaściwy plik, a zapomnieliś o zatwierdzeniu innego.


```

MINGW64; c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   TODO.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        DONE.txt
        fileforgotten.txt
        filenottocommit.txt

no changes added to commit (use "git add" and/or "git commit -a")
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |

```

Rysunek 5.8. Zmodyfikowane i nieśledzone pliki zostały wyróżnione

```

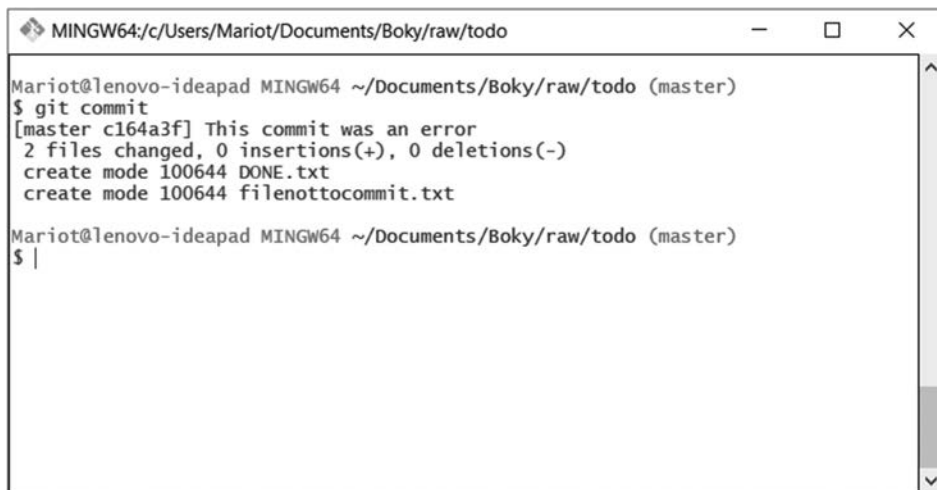
MINGW64; c/Users/Mariot/Documents/Boky/raw/todo
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#   new file:   DONE.txt
#   new file:   filenottocommit.txt
#
# Changes not staged for commit:
#   modified:   TODO.txt
#
# Untracked files:
#   fileforgotten.txt
#
~
~
~
~
~
~
~
~
<ot/Documents/Boky/raw/todo/.git/COMMIT_EDITMSG [unix] (23:55 12/08/2019)1,1 All
-- INSERT --

```

Rysunek 5.9. Komunikat commita jest ostatnim zabezpieczeniem

Przed wszystkim należy usunąć ostatni commit z projektu poleceniem `git reset`. Użyjemy opcji `--soft`, aby wprowadzone wcześniej zmiany pozostały obecne w katalogu roboczym. `HEAD~1` informuje, że poprzedni commit wykonany na `HEAD` wskazuje aktualny.

```
$ git reset --soft HEAD~1
```



```

MINGW64:c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git commit
[master c164a3f] This commit was an error
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 DONE.txt
create mode 100644 filenottocommit.txt
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ |

```

Rysunek 5.10. Opis commita. Coś sknociliśmy

Możesz następnie wycofać plik z przechowalni, używając ponownie `git reset`:

```
$ git reset HEAD filenottocommit.txt
```

Sprawdź, czy polecenia zadziałały zgodnie z zamysłem. W tym celu rzuć okiem na aktualny status projektu.

```
$ git status
```

Otrzymasz wynik taki jak na rysunku 5.11.

Jak widzisz, *filenottocommit.txt* nie jest teraz śledzony, ponieważ usunęliśmy go z przechowalni. Oczywiście *fileforgotten.txt* też nie jest śledzony, gdyż nie wysłaliśmy go do przechowalni. Znajduje się w niej jedynie *DONE.txt*, bo nie ruszaliśmy tego pliku po zatwierdzeniu.

-
- **Ostrzeżenie** Używaj polecenia `reset` ostrożnie. Jest bardzo niebezpieczne. Koniecznie dokładnie sprawdzaj, co napisałeś.
-

Następnie wyślij do przechowalni właściwy plik:

```
$ git add fileforgotten.txt
```

Skoro już wysłałeś właściwe pliki, możesz zatwierdzić projekt:

```
$ git commit
```

Zamieść w opisie commita jakiś błąd. Przyda się to przy zapoznawaniu się z kolejną funkcją Gita.

```

MINGW64;c/Users/Mariot/Documents/Boky/raw/todo
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   DONE.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   TODO.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    fileforgotten.txt
    filenottocommit.txt

Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/todo (master)
$

```

Rysunek 5.11. Status projektu po resecie

Poprawianie commita

W przypadku zwykłych błędów takich jak pomyłka w opisie commita nie ma potrzeby modyfikowania całego commita. Wystarczy go poprawić. Wypróbujmy to na naszym projekcie!

```
$ git commit --amend
```

Poprawianie wygląda tak samo jak zatwierdzanie zwykłego commita, z tym że opis commita już jest gotowy (rysunek 5.12).

Możesz dowolnie zmienić opis, a następnie standardowo zapisać plik i wyjść z edytora.

To takie proste! Spójrz na nazwę nowego commita i porównaj ją z nazwą starego. Są różne. Jest tak, ponieważ nazwa commita powstaje w wyniku haszowania informacji zawartych w migawce. Różne stany projektu przekładają się zatem na różne nazwy.

Jeszcze jedna, ostatnia uwaga na temat modyfikowania commitów: nie nadużywaj tej możliwości! Owszem, błędów w kodzie należy się wystrzegać i na ogół dążymy do natychmiastowego ich usuwania. Błędy też jednak pomagają nam się doskonalić; śledzenie własnych błędów sprzyja nauce.

Ćwiczenie: staranne poprawianie commita

Wróć do listy zadań do wykonania. Celem tego ćwiczenia jest staranne poprawienie commita.

- Edytuj kilka plików i wyślij je do przechowalni.
- Zatwierdź je i popętnij błąd w opisie commita.
- Wycofaj plik z przechowalni.
- Wyślij inny plik do przechowalni.
- Popraw commit i wprowadź właściwy opis.

```

MINGW64:c/Users/Mariot/Documents/Boky/raw/todo
This commit was an error
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Tue Aug 13 00:28:47 2019 +0200
#
# On branch master
# Changes to be committed:
#   new file:   DONE.txt
#   new file:   fileforgotten.txt
#
# Changes not staged for commit:
#   modified:   TODO.txt
#
# Untracked files:
#   filenottocommit.txt
#
~
~
<ot/Documents/Boky/raw/todo/.git/COMMIT_EDITMSG [unix] (00:36 13/08/2019)1,1 All
ariot/Documents/Boky/raw/todo/.git/COMMIT_EDITMSG" [unix] 18L, 413C

```

Rysunek 5.12. Edycja opisu commita

Podsumowanie

Ten rozdział był poświęcony głównie poruszaniu się po różnych wersjach projektu oraz ich wycofywaniu i poprawianiu. Nie powinieneś mieć już problemów z wprowadzaniem drobnych poprawek do commitów. Koniecznie przeczytaj jeszcze raz pierwszy podrozdział, ponieważ znajdują się w nim informacje kluczowe dla wszystkiego, co robisz w Gicie. Gdy ktoś Cię obudzi w środku nocy, musisz być w stanie wyjaśnić różnice między trzema stanami Gita.

Kolejny rozdział jest krótki, ponieważ poruszymy w nim wyłącznie teorię. Dowiesz się, jak napisać schludny opis commita, co uwzględniać, a co ignorować w commitach, a także poznasz błędy, które najczęściej popełniają początkujący. Koniecznie przeczytaj następny rozdział uważnie, ponieważ zawarta w nim wiedza istotnie pomoże i Tobie, i Twojemu zespołowi. Naprzód!

PROGRAM PARTNERSKI

— GRUPY HELION —

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

GitHub. Odzyskaj kontrolę nad wszystkimi wersjami Twoich projektów!

Rozbudowane projekty wymagają zaangażowania zespołu programistów. By ich współpraca była efektywna, niezbędny jest system kontroli wersji, taki jak Git. Umożliwia on tworzenie historii projektu, zarządzanie jego wersjami, przeglądanie wszystkich zmian, a także przywracanie pliku do dowolnej wcześniejszej wersji. Repozytoria Gita można przechowywać w GitHubie — w ten sposób bez ponoszenia dodatkowych opłat udostępnia się kod innym osobom. Korzystanie z Gita i GitHuba nie jest skomplikowane, wymaga jednak pewnej wiedzy i wprawy.

To książka przeznaczona dla programistów, którzy chcą zacząć pracę z Gitem i GitHubem. W każdym rozdziale zawarto wyłącznie przydatne informacje, a te uzupełniono licznymi ćwiczeniami. Dzięki temu równocześnie możesz się uczyć Gita i nabierać sprawności w posługiwaniu się tym systemem. Przewodnik podzielono na trzy części tematyczne dotyczące kontroli wersji, zarządzania projektami i pracy zespołowej. To ułatwi Ci wdrożenie się do rzeczywistej pracy i rozwiązywanie problemów. Poznasz zasady planowania i realizacji projektów z GitHubem, a także wypróbujesz sposoby rozstrzygania konfliktów scalania, co sprawi, że poczujesz się pewniej w pracy zespołowej w profesjonalnym środowisku.

W książce:

- czym są, do czego służą i jak działają systemy kontroli wersji
- jak przygotować Git do pracy i jak ją rozpocząć
- obieg pracy w GitHubie: zgłoszenia, gałęzie, pull requesty
- konflikty scalania, ich rozstrzygnięcie i zarządzanie zmianami kodu
- najprzydatniejsze narzędzia GUI Gita
- zaawansowane polecenia Gita i rozwiązywanie częstych problemów

Mariot Tsitoara programuje w Pythonie i JavaScriptcie. Od 2015 roku jest przedstawicielem Mozilli. Często uczestniczy w konferencjach technicznych, na których wypowiada się na temat otwartego kodu i nowych technologii, takich jak Rust, WebVR i WebAssembly. Mieszka w Bordeaux, zajmuje się tworzeniem kodu niewielkich specjalistycznych narzędzi edukacyjnych.

Helion 	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI <i>Sięgnij po więcej!</i> 	
 helion.pl	SZKOLENIA 	ISBN 978-83-283-8735-5	
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	AKADEMIA IT & BUSINESS HELIONSZKOLENIA.PL	 9 788328 387355	
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 69,00 zł	

Apress®